On the Expressiveness of Purely Functional I/O Systems

Paul Hudak
Raman S. Sundaresh

Department of Computer Science
Yale University
Box 2158 Yale Station
New Haven, CT 06520

March 10, 1989

Abstract

Functional programming languages have traditionally lacked complete, flexible, and yet referentially transparent I/O mechanisms. Previous proposals for I/O have used either the notion of *lazy streams* or *continuations* to model interaction with the external world. We discuss and generalize these models and introduce a third, which we call the *systems* model, to perform I/O. The expressiveness of the styles are compared by means of an example. We then give a series of surprisingly simple translations between the three models, demonstrating that they are not as different as their programming styles suggest, and implying that the styles could be mixed within a single program.

The need to express non-deterministic behavior in a functional language is well recognized. So is the problem of doing so without destroying referential transparency. We survey past approaches to this problem, and suggest a solution in the context of the I/O models described.

The I/O system of the purely functional language HASKELL is presented. The system includes a rich set of operations, and distinguishes between *file* and *channel* I/O. The approach to non-determinism is also presented. A useful aspect of the design is that it includes a rigorous specification of the behaviour of the operating system, thus precisely fixing the semantics of the various I/O operations. The HASKELL I/O system is capable of supporting many other paradigms of concurrent computation in a natural way. We demonstrate this through the emulation of Actors, UNITY, CSP, CCS and Linda.

1 Introduction

If functional languages are to be used in real applications programming, an effective I/O system seems essential. To many, however, the mention of I/O conjures up an image of state, side-effects, and sequencing. Is there any hope of achieving ***purely functional***, yet ***universal***, and of course ***efficient*** I/O? This is the question we will address in this paper.

To begin, we identify three basic requirements for such an I/O mechanism:

- *Referential Transparency.* We consider it inappropriate to use a side-effecting function which returns the next value from the input stream, because such a function (call it `get_char`) would destroy *referential transparency* (`get_char` would have to return a different value every time it is called). Similarly, a side-effecting print function (call it `put_char`) would destroy referential transparency because, for example, "`f (put_char 'a') (put_char 'a')`" would not be the same as "`f x x where x = put_char 'a'`."

- *Efficiency.* An essential property of an I/O scheme is efficiency. It must be possible to implement the scheme efficiently without resorting to expensive operations like taking "snapshots" of the system state.

- *Co-operation.* Another requirement for a practical I/O scheme is the ability of the external world to observe the effects of a program even though the program has not yet terminated. Conversely, a program should also be able to observe changes made to the system state after the program has begun. Thus we must view with suspicion, for example, any scheme which maps a single initial state into a single final state.

The last requirement is important for several reasons. The first is that we anticipate writing programs which co-operate interactively with traditional operating system services. This includes traditional file manipulation (for example the changes made to a file by an editor need to be visible to other programs without terminating the editor), device handling (for example printing), and in general communication with other programs, machines, or users (for example standard input/output and TCP/IP protocols).

Another reason is that we do not want a non-terminating (or just very slow) program to tie up computer resources; we would naturally expect the I/O effects of various programs running under a single operating system to be interleaved. In other words, suppose `prog1` and `prog2` are two programs where `prog1` performs the I/O operations `op1` and `op2` and `prog2` performs operations `op3` and `op4`. If `os` is the operating system which maps a list of programs and an initial state into a final state, we would expect:

```
    os [prog1, prog2] initial_state
```

to be the final state resulting from an arbitrary interleaving of `op1`, `op2`, `op3` and `op4`. Thus for each I/O model that we propose, we will either write such an operating system `os` to show that the above property does indeed hold, or show why such an operating system cannot be written.

1.1 Purely Functional I/O

I/O proposals meeting all of our requirements are rather difficult to come by in existing functional languages. ML [HMT88] and ALFL [Hud84], for example, use side-effecting primitives to do I/O. Since ALFL uses normal order evaluation, it also provides ways of forcing execution order to ensure deterministic results. We summarize other existing approaches below.

Streams. An elegant and popular model that goes a long way toward meeting our requirements is the use of *streams*, lazy lists of data objects. The name *stream* was first coined by Landin [Lan65]; since then, several functional languages have used streams for I/O, including Ponder, Hope, and Miranda[1] [Fai85,BMS80,Tur85]. In these languages predefined identifiers are typically provided which are bound to specific I/O channels. For example, the stream of input characters from the keyboard might have the name `kb`, and the stream of output characters to the display might have the name `display` — the operating system will provide the binding for `kb`, and the program is expected to provide the binding for `display`.

Although elegant, there are at least three problems with these previous uses of the stream idiom:

1. They are not completely general, since typically the I/O devices and the operations on them are pre-determined and fixed into the language.
2. The semantics of interactive I/O (for example, interaction with the user) is not entirely clear (in some languages this problem is admitted up front by providing a mechanism to control the order in which the streams are consumed and produced).
3. Anomalous situations are ignored; the possibility of error is generally not accounted for.

In the stream model of I/O that we present in Section 2.1, streams are used to invoke arbitrary I/O operations with arbitrary responses (typically either success or failure). In addition we retain the elegant use of streams to model interactive I/O, but we are careful to precisely define the semantics such that the input from a user, for example, can depend on output from the program (characteristic of many interactive applications). All three of the requirements stated earlier are satisfied.

Continuations. Another model of I/O, namely *continuation* based I/O, was first proposed in the context of a functional operating system called Nebula [Kar81]. The idea reappeared in a parallel functional language called PFL [Hol83], which is based on CCS [Mil81], and more recently was adopted in the functional language Hope [MH88]. The continuation model is characterized by a set of *transactions*, which are functions that typically take a *success continuation* and *failure continuation* as arguments; these continuations are in turn functions that generate more transactions. The continuation model is appealing because it appears to be quite general, and the continuation structure makes it easy to reason about the sequentiality of the induced effects.

In Section 2.2 we present a continuation model very similar to these, with one important exception: we assume a non-strict (i.e. lazy) functional language, whereas previous languages using the continuation approach have been strict. This not only simplifies the design, but in addition allows us to use the interactive lazy stream idea *within* the continuation model. More specifically,

[1]Miranda is a trademark of Research Software Ltd.

in PFL and Hope individual characters are read by each continuation operation, while in our model a single read operation returns a lazy stream. Thus we are able to combine the virtues of both "idioms" – streams to model demand-driven sequences of data, and continuations to enforce control restrictions.

Systems. We introduce a third model of I/O in Section 2.3, which we call the *systems* model,[2] in which I/O is viewed as a series of transformations to an initial "system" that captures the state of the operating system. One by-product of investigations into this model is that we cannot view a functional program as a function from a *single* initial state to a *single* final state (at least not without severely restricting functionality). Thus our model actually uses a stream of systems, but surprisingly, *only as output*; a single initial system is sufficient to meet all of our requirements.

It is worth noting that the language FL [BWW86,WW88] performs I/O by adding a *history* parameter to every function, both as argument and as result, and the notion of a history bears strong resemblance to our notion of a system. However, FL is a strict language, and the history objects are *implicit*. The strict semantics allows the designers to cleanly define the order in which I/O occurs, and ensures that exactly one history object is in existence at any given time. The situation is not as simple with a non-strict language, and we view the use of implicit state variables as undesirable because of its "imperative feel."

1.2 Equivalence of Models

These three models of I/O — streams, continuations, and systems — induce very different programming styles on the user. However, it turns out that there exist surprisingly simple definitions of any one of the models in terms of either of the other two; we give such translations in Section 2.4. The existence of such translations has two important consequences: First, it indicates that these models are indeed equal in "expressivness," thus ending long-standing debates over this issue. Second, and perhaps more importantly, a language designer can provide all three styles within the same language, simply by choosing one as "primitive" and then providing modules that define each of the others in terms of the first (i.e. the styles do not have to be wired into the language). A programmer is free to choose the style most suitable to a particular application, and even intermix several styles in a single program.

1.3 Expressing Non-Determinism

Non-determinism is a pervasive property of real systems, and thus it seems desirable to express non-deterministic behavior in a functional program. For example, if a program is to service two independent sources of input (say two keyboards), then any fixed interleaving of the two streams is unsatisfactory; what is needed is a non-deterministic merge of the two streams. Another use of non-determinism is to express the so-called *parallel-or* function (which has the property that `or` $\perp$ `True = or True` $\perp$ `= True`).

[2]Such a model has been part of the folklore for some time, and was an active topic of the HASKELL committee, but we have never seen the details worked out as we have done here.

The naive addition of a **merge** or **amb** operator to express such behaviors is undesirable because of the loss of referential transparency. Also, the semantics and expressive power of such operators are far from clear; Clinger [Cli82] discusses several complications caused by their introduction.

In Section 3 we review past approaches language designers have taken in adding non-deterministic behavior to a functional language, and then suggest a solution based on insights obtained from the design of our I/O systems. The result is a form of non-determinism that maintains referential transparency *within any single program*, but not within a collection of programs; we argue that this is a practical compromise solution to the problem.

1.4 HASKELL I/O

To show how all of these concepts look when incorporated into a real functional language, we describe in Section 4 the design of the HASKELL I/O system which combines at least some of the ideas we present.[3] We demonstrate the power of HASKELL I/O by showing how it can easily and naturally simulate various other concurrent programming paradigms, including Actors, UNITY, CSP, CCS and Linda.

(In the remainder of this paper all functional programming examples will be written using HASKELL syntax.)

2 Three Models for Purely Functional I/O

We will introduce our three models of I/O — streams, continuations, and systems — by way of a slightly modified version of an example from Kernighan and Ritchie's *The C Programming Language* [KR78]. This example is a simple file display program which prompts the user for a number of file names, and reads and displays their contents.

The imperative version in C is shown below. In this program the procedure **get_token** reads the next token off of an input stream. In the functional programs to be given shortly, **get_tokens** is a function which tokenizes a character stream. The functional models of I/O assume that characters typed in at a keyboard are not echoed by the operating system, whereas in the C program they are.

[3]For a complete specification of the HASKELL I/O system, see [Hea88].

```
#include <stdio.h>
main () /* cat: concatenate files */
{
   FILE *fp, *fopen();
   char *get_token();
   char *name;

   printf("type in file names\n");
   while (name = get_token(stdin))
      if ((fp = fopen(name,"r")) == NULL) {
           printf("can't open file\n");
   break;
      } else {
   file_display(fp);
   fclose(fp);
      }
}

file_display(fp) /* copy file to standard output */
FILE *fp;
{
   int c;

   while ((c = getc(fp)) != EOF)
     putc(c, stdout);
}
```

2.1 The Stream Model

In our stream model of I/O a program is simply viewed as a black box that generates a stream of I/O *requests*; these requests are given to the operating system, processed one-by-one, and returned to the program as a stream of *responses.* Thus if **Request** is the datatype of requests, and **Response** is the datatype of responses, then a program p has type **[Response] -> [Request]**; the nth request generates the nth response.

The response itself depends on the request, and would normally include the possibility of error. However in the sample program below, tests for failure are omitted, as they are in the C program. (In HASKELL a program has value main.[4])

[4]Note that **resps** cannot be taken apart by pattern matching, since this would entail evaluating the response list before any requests have been issued, resulting in $\perp$.

```
main resps =
    [ AppendChannel "stdout" "type in file names\CR\",
      ReadChannel "stdin" ]
    ++ file_display (tl (tl resps))
                    (get_tokens
                       (case resp!!2 of
                             Return user_input -> user_input))

    where file_display resps [] = []
                         resps (name:names) =
              [ AppendChannel "stdout" name,
                ReadFile name,
                AppendChannel
                  "stdout"
                  (case resps!!2 of
                     Failure msg -> "can't open file\CR\"
                     Return file_contents -> file_contents) ]
              ++ file_display (tl (tl (tl resps))) names
```

(The functions **hd** and **tl** are the obvious "head" and "tail" functions defined on lists; similarly, we will use **fst** and **snd** for selectors on tuples.) Referential transparency is preserved because there is no lexical connection between a request and a response. Thus although two different "read" requests of the same file are equivalent *values*, their effect will depend on their position in the request list, and each could invoke a *different* response, reflecting the fact that other programs may have modified the file in question between the two reads.

We now present an operating system function **os** which services the requests of (for simplicity) two programs **p1** and **p2**, and executes them in an interleaved manner. This will demonstrate the model's capability for co-operation/communication among different programs engaged in I/O.

```
(tagged_responses, final_state) = os tagged_requests initial_state
where tagged_requests = merge requests1 requests2
      requests1 = p1 (untag 1 tagged_responses)
      requests2 = p2 (untag 2 tagged_responses)
```

os maps a list of tagged requests and an initial state of the system into a list of tagged responses and a final state of the system. The non-deterministic interleaving of effects is accomplished via the **merge** operator which produces a tagged non-deterministic merge of its two list arguments (we return to the issue of non-determinism in Section 3). **untag** picks out responses of a given tag value from a tagged list of responses. Note that this scheme can be generalized to an arbitrary number of programs.

The state of the system is a single-threaded object, and so can be implemented efficiently. On the other hand, a certain amount of file copying may be necessary. For example, consider the following scenario: First a program issues a read file request, but since the read is done lazily, no actual reads are done until the value of the file is demanded. Suppose in the meantime another program updates the file. This would mean that a copy of the old file must be saved until the reference to it is "released" by the first program.

2.2 The Continuation Model

In this model, instead of having to manipulate lists of requests and responses, the programmer initiates I/O operations via continuation style transactions. For example, a request such as **ReadFile name** in the systems model corresponds to the transaction:

```
ReadFile name (msg -> error_transaction)
              (contents -> success_transaction)
```

where the second and third arguments are the failure and success continuations, respectively. The value of the overall expression is the error continuation applied to the resulting error message if the read fails, and is the success continuation applied to the contents of the file if the read succeeds. Below we show the running example in this style. Note that it is laid out in such a way as to facilitate an "imperative reading" of the program.

```
main = AppendChannel "stdout" "type in file names\CR\" exit (
       ReadChannel "stdin" exit (user_input ->
       file_display (get_tokens user_input)))

       where file_display [] = done
             '            (name:names) =
                 AppendChannel "stdout" name exit (
                 ReadFile name
                     (msg -> AppendChannel "stdout"
                                           "can't open file\CR\"
                                           exit
                             Done)
                     (contents ->
                 AppendChannel "stdout" contents exit (
                 file_display names)))
exit = msg -> Done
```

Referential transparency is maintained in this model for the same reason as in the stream model: there is no lexical connection between transactions and results. However, whereas in the stream model the functional connection was via position in a list, here the connection is established by way of continuations.

We will write the operating system function for this model too, to show how programs indulging in this form of I/O can co-operatively change a shared state.

```
os p1 p2 state = os' p1' p2'
   where os' Done Done = state
         os' (<transaction> <args> succ_cont fail_cont) p =
             os (case resp of
                      Return result -> succ_cont result
                      Failure msg   -> fail_cont msg   ) p state'
             where (resp, state') = process <transaction> <args> state
         os' p1 p2 = os' p2 p1
         (p1',p2') = perm p1 p2
```

Here the non-deterministic interleaving of effects is accomplished via **perm**, which returns a non-determinstic permutation of its two arguments, and is "bottom-avoiding" in its first argument. The remainder of the definition should be self-explanatory.

Since at any one time, only one version of the system state is in existence, this model can be implemented efficiently. However the comment made about file copying in the stream model applies equally here.

2.3 The Systems Model

The naive version of the systems model of I/O views a program simply as a function from an initial system to a final system, where a "system" is meant to capture the entire operating state of interest (files, devices, etc.). Normal I/O operations like reading and writing files are functions which take a system and return a result and an "updated" system.

This view, while enticing, is not workable.[5] To see this, let us try to write an operating system function for two programs doing I/O in this model.

```
os p1 p2 state = amb (p1 (p2 state)) (p2 (p1 state))
```

where *amb* non-deterministically returns either of its arguments. There is not much more than this that we can do! It is apparent that there can be no interleaving of the effects of the two programs.

Ignoring this problem for the moment, let us look at the model from the angle of efficiency. The fact that the I/O operations of two programs cannot be interleaved means that all the operations desired by a program must be performed contiguously. Since the model makes it possible for non-single-threaded usage of the system objects, expensive simulation of I/O operations has to be done within the program. This can be made clear by the following example: a program that writes a file "foo" and returns the original system if the write succeeds.

```
prog = sys0 -> case resp of
                 Success     -> sys0
                 Failure msg -> sys1
       where (resp,sys1) = writefile "foo" "junk" sys0
```

[5]Thomas Johnsson first pointed out this flaw in a translation of this model into the streams model.

In this program the value of `resp` is needed to determine which system to return; i.e. the program requires the result of the `writefile` even before "returning" anything to the I/O system. Thus the write must be simulated within the program. This means that a proper implementation of this model is going to be arbitrarily inefficient.

An obvious fix to this problem is a model in which a program is viewed as a function from a *list* of systems to a *list* of systems (an alternative considered by the HASKELL committee), where the effect of an operation on one system is visible only in the next input system. This model has an efficient implementation since the system state is single-threaded — the order in which I/O operations are carried out is fixed by the output list of systems. It is also clear that the model can interleave the effects of many programs.

Surprisingly, however, this same functionality can be obtained by a model where programs map a *single* initial system to a *list* of output systems! We prefer this latter model because it is equivalent in power to the former, but is more convenient to use in that it relieves us from the tedium of managing two separate lists. As before, the result of an I/O operation on a system is a response and a modified system; the new system can then be used for further I/O. The main advantage over the stream model is that some of the tedium of matching requests and responses is avoided. We show the running example in this style:

```
main sys0 =
  sys1 : sys2 : file_display sys2 (get_tokens
                                     (case resp2 of
                                        Return user_input -> user_input))

  where (sys1,resp1) = appendChannel "stdout" "type in filenames\CR\" sys0
        (sys2,resp2) = readChannel "stdin" sys1
        file_display sys [] = []
        '            sys (name:names) =
          sys1 : sys2 : sys3 : file_display sys3 names
          where (sys1,resp1) = appendChannel "stdout" name sys
                (sys2,resp2) = readFile name sys1
                (sys3,resp3) =
                   appendChannel
                     "stdout"
                     (case resp2 of
                        Failure msg -> "can't open file\CR\"
                        Return file_contents -> file_contents)
                     sys2
```

Although we have not yet specified the internals of a "system" object, in the next section we give a representation of a system using streams. One implication of this translation is that the interleaving property we showed for the streams model holds for the systems model too.

Finally, we note that the sequence of intermediate systems is clearly specified, which means that the order in which I/O operations are to be done is fixed. This means that the system state is single-threaded, which in turn implies that efficient implementations are possible.

2.4 Equivalence of the Three Models

Although it is not apparent at first sight, there exist simple translations of one style of I/O into another. In the sections that follow, we show how each model can support the other two.

2.4.1 Streams as Primitive

Given that a program is a function from responses to requests, how can we support the other two styles? The continuation style can be supported simply by defining each transaction to be a function from responses to requests, just like a program in the streams model. For example, the definition of `readChannel` and `done` would be:

```
readChannel name fail_cont succ_cont =
  resps -> ReadChannel name :
            case (hd resps) of
              Failure msg     -> fail_cont msg      (tl resps)
              Return contents -> succ_cont contents (tl resps)
done resps = []
```

Given these definitions, a program written in the continuation style will map a list of responses to a list of requests as desired.

The system style of I/O can also be supported by representing a system as a response list-request pair:

```
type System = ([Response],Request)
```

The I/O operations are then defined to pull out a response and return a new system. As an example consider the definition for `readChannel`:

```
readChannel name (resps, req) =
  (hd resps, (tl resps, ReadChannel name))
```

Given these definitions, a program `p` written in the systems model described earlier can be coerced into the streams model as follows:

```
resps -> map snd (p (resps, NullRequest))
```

where `NullRequest` is a dummy request.

2.4.2 Continuations as Primitive

Using continuations as primitive, first we will show how to provide a stream style of I/O. Given a program written in the stream model, its meaning is given by the following continuation style function `c`; which can be viewed as an interpreter for the stream model, written in the continuation model.[6]

[6]A translation similar to this was first given by Simon Peyton Jones.

```
c prog =
  case (prog bottom) of
    [] -> Done
    (ReadChannel name) : reqs ->
       readChannel name
                  (msg      -> c (resps -> tl (prog (Failure msg      : resps))))
                  (contents -> c (resps -> tl (prog (Return contents : resps))))
    ....and similarly for each other request....

bottom = bottom
```

It is interesting to note that this translation has an unavoidable "space leak," in that the two continuation arguments grow linearly in the number of requests issued. This also implies that the time to process *n* requests grows as n^2, since the *nth* request can only be obtained by applying **tl** *n-1* times. The space leak exists because the function has to be reevaluted to get information about the rest of the program.[7]

A very similar translation provides the systems style of I/O, with systems and the I/O functions being represented as in the previous section. The details are omitted.

2.4.3 Systems as Primitive

Using the systems model, simulating streams is rather straightforward. Consider a program **s** written in the streams model; to run it in the systems model, we would use the following "interpreter:"

```
sys0 -> map fst answers
  where answers = scan (x -> y -> (req-to-fn y) (fst x))
                       (sys0,NullResp)
                       requests
        requests  = s responses
        responses = map snd answers
```

req-to-fn gives the system operator corresponding to a request. For example, "**req-to-fn (ReadChannel name)**" is "**readChannel name**." **scan** is defined by:

```
scan f a [] = []
'    f a (x:xs) = f a x : scan f (f a x) xs
```

To provide a continuation style, we write the transactions in the systems model. For example, the **readChannel** and **done** transactions are defined by:

[7]Given **f** $\perp$ **= a:**$\perp$, the second request can be accessed only by **hd(tl(f a:**$\perp$**))**, i.e. **f** must be reevaluated on the whole argument. Compare this with a program in the continuation model: Given a program **<transaction1> <success continuation> <failure continuation>** all we have to do to get at the second transaction given the response to the first one, is to apply one of the continuations (depending on whether the operation succeeded or failed) to the response.

```
readChannelC name fail_cont succ_cont =
  sys -> sys' : (case resp of
                   Failure msg     -> fail_cont msg      sys'
                   Return contents -> succ_cont contents sys')
  where (sys',resp) = readChannelS name sys

done = sys -> []
```

where `readChannelC` and `readChannelS` are the continuation and system functions, respectively, for reading a channel. Thus, given the above definitions, a program written in the continuation style has type `System -> [System]`.

2.5 Comments

Given that the models are all equivalent, what can we say about their merits with respect to other factors such as style, ease of programming, etc.? In this section we comment on these and other issues.

- Programs using the stream style must be written with care, since subtle strictness bugs can arise. The root of the problem is that examining a response "before" the corresponding request is issued results in "deadlock" (i.e. $\perp$), which is an easy mistake to make since the response list and request list are completely separate. A particularly subtle form of this bug occurs in combination with pattern matching. For example, the following program, which purports to write a simple message to standard output, will not work.

  ```
  main [resp] = [ WriteChannel "stdout" "hello" ]
  ```

  This is because pattern-matching demands that the response list be of a certain structure even before the simple request is issued.

- The equivalence of the styles has an interesting implication: they can be used together in a single program, especially (for stylistic reasons) when the program is divided into modules. Each module can use the I/O style most suited to it or most preferred by its author.

- All three I/O models benefit from non-strict semantics in two distinct ways. The first is that the operating system can delay responding to an I/O operation until the response is demanded by the program. To see how this is useful consider the following compiler scenario: For each library function mentioned in a source program, a certain information file must be read. Ordinarily, one must choose either to make an extra pass of the program to pick out the library functions and read the required information, or one must interleave code to do the reading with the main pass of the compiler. Both alternatives are unsatisfactory; the first is inefficient, while the second destroys modularity. In our I/O systems lazy evaluation comes to the rescue. We would first "read" the required information for all the library files; but the actual reading takes place only when the values are demanded, i.e. only for library functions mentioned in the program. Thus both efficiency and modularity are preserved.

The second benefit is that since the response to requests like `ReadChannel` is a lazy stream itself (i.e. a stream of characters), the routines which do the computation need never deal with I/O or synchronization of I/O; they simply get a character list as argument, which they process in the usual functional style.

- An important characteristic of all the functional I/O schemes discussed is their ability to "feel" external effects. This capability is invaluable for interactive programs like editors. The streams and continuation models obviously provide this capability, but what is perhaps surprising is that the systems model does also, even though it takes only one initial system as input. This can be seen by the simulation of systems using streams given earlier.
- In our experience with writing example programs, we have found that the continuation style is often easier to use and the resulting programs easier to read. The reasons for this are that the continuation model reduces the syntax for handling the responses to I/O operations; the response handling is in a sense "built-in."

3 Non-Determinism

One of the well known shortcomings of functional languages is their inability to express non-deterministic behaviour. Henderson [Hen82] shows how introduction of such behavior into a functional language makes it possible to write a wide range of useful operating system-like programs. He introduces non-determinism by the use of an operator called `merge`, which produces the non-deterministic merge of two lists. The problem with this solution is that referential transparency is destroyed. This implies that equational reasoning, an important program verification tool, can no longer be used. Also, programs using these techniques tend to be difficult to read. The term "sphagetti programming" has been used to describe them [Sto84,Tur87]. The explicit use of non-determinism also raises a host of questions about its interaction with the parameter passing mechanism [Cli82] and the formal semantics of non-deterministic operators is complex [SS88].

First, we shall critically survey some of the proposals made to overcome these problems, and then outline our proposal.

3.1 Stoye's Approach

Stoye [Sto84] views an operating system as a collection of *processes*, each of which is a functional program with a single input list and a single output list. The output list of a process is a list of tagged data, the tag specifying the addressee. The non-determinism in the system comes into play when two processes send messages to the same process. All the messages sent to a given process are "merged" into a single list and given to it. This merge occurs outside of all the processes in what Stoye calls the *sorting office*. The advantage of this scheme is that each of the processes themselves are referentially transparent, and can be subjected to equational reasoning. Turner [Tur87] refined this idea somewhat for use in another functional operating system effort, the KAOS project.

Stoye applies this style profitably to the task of writing operating system programs like device handlers. But we claim that there are applications where this style is not suited ideally. Consider the well known *generate-and-test* paradigm: first compute a set of candidate solutions to a problem, then apply in parallel a test to each of them. We wish to be informed of solutions as and when they are found. In a functional language without non-determinism, this behaviour is impossible to obtain. The list of test results is obtained in a fixed order and if say, any one of the tests takes a long time, solutions further down in the list which are already available will be delayed. In the case that a test does not terminate, solutions further down the list may never be displayed.

To express this behaviour in Stoye's scheme, we would need to create a separate process for each of the tests and have them send their results to another process which collects them — the non-determinism in the sorting-office gives the desired behaviour. But now the code is considerably more complex, and modularity has been impaired.

3.2 Burton's Approach

Burton [Bur88] addresses the problem of loss of referential transparency in a different way, and comes up with a solution involving what he calls "pseudo-data." He proposed to supply each program with an extra argument: a infinite binary tree of values. Whenever the program needs to make a non-deterministic choice, the binary tree is consulted (as a kind of oracle). A tree is chosen rather than a list because any number of subtrees can be extracted from a binary tree. Burton notes that in practice the values in the tree will be determined at run-time (when used as an argument to a special function), but once fixed will never change.

While Burton's proposal does provide non-determinism with referential transparency, it still advocates unfettered (and possibly undisciplined) use of the `merge` operator. Stoye [Sto84] and Turner [Tur87] claim that the unrestricted use of a non-deterministic operator like merge reduces the readability of the program; in Burton's proposal this criticism is heightened because the "pseudo-data" tends to clutter the program even more.

3.3 Our Approach

We will refer to Burton-style non-determinism as being "amb-like," and to Stoye-style non-determinism as being "process-like." In this section we will show how both kinds of non-determinism may be achieved via small extension to our I/O models, and we discuss a related method used in HASKELL.

A simple way to achieve amb-like non-determinism in the streams model is to introduce a request called `amb` which takes two arguments, and the corresponding response would return the non-deterministic choice between those arguments. The same idea could be used with either the continuation or systems model as well. In this way we can achieve amb-like non-determinism in a referentially transparent way, but without cluttering up a program with "psuedo-data." The "generate-and-test" problem mentioned earlier can be solved nicely with this approach.

Similarly, the process-like non-determinism that Stoye uses can be achieved by providing a special request to perform the function of Stoye's sorting office. In a later section we will in fact exploit this idea in demonstrating how to express other paradigms of concurrency.

As can be seen, the fundamental idea behind our approach to non-determinism is the use of the operating system to provide "special non-deterministic services," such as `amb`, the sorting office, or whatever. Another example of such specialization is the non-deterministic servicing of multiple agents (such as two keyboards), which can be handled by generalizing the `ReadChannel` request to one that takes a *list* of channels, the response being a non-deterministic `merge` of the streams. This was in fact the solution adopted in HASKELL.

Analogous to channel read, the channel write request `WriteChannel` could also be generalized to take a number of streams and write their non-deterministic merge to the named channel. This, however, may require logical parallelism *within* a program, and thus may be more difficult to implement than any of the ideas mentioned so far. Nevertheless, it may be desirable as an alternative solution to, for example, the "generate-and-test" paradigm.

4 The HASKELL I/O System

This sections describes in detail the specification of the HASKELL I/O system. Of noteable interest is section 4.6, where the semantics of a HASKELL program engaged in I/O is described within the operating system in which it runs.

The HASKELL I/O system unifies two popular styles of purely functional I/O processing: the *stream* model and the *continuation* model. Programs in either style may be combined under this framework with a well-defined semantics. The specific I/O operations available in each style are identical; what differs is the way they are expressed. In both cases arbitrary I/O operations within conventional operating systems may be induced while retaining referential transparency internal to a program.

4.1 Stream-based I/O

A HASKELL program engaged in input/output (I/O) processing must have type:

```
Behavior = [Response] -> [Request]
```

Intuitively, `[Response]` is an ordered list of *responses*, and `[Request]` is an ordered list of *requests*. The nth response is the reply of the operating system to the nth request.

The required requests for a valid implementation are:

```
data Request = ReadFile Name
             | WriteFile Name Contents
             | AppendFile Name Contents
             | DeleteFile Name
             | ReadChannel Name
             | ReadChannels [Name]
             | AppendChannel Name Contents
type Name = String
```

Requests operate on two conceptually different components of a system: a *file system* (responding to the first four requests above), and a *channel system* (responding to the last three). The file system is fairly conventional: a mapping of file names to contents. The channel system consists of a collection of *channels*, examples of which include standard-input and standard-output. A channel is a one-way communication medium – it either consumes values from the program (via `AppendChannel`) or produces values for the program (via response to `ReadChannel` or `ReadChannels`). Channels communicate to and from *agents* (a concept to be made more precise later). Examples of agents include line printers, disk controllers, networks, and human beings. As an example of the latter, the *user* is the consumer of standard-output and the producer of standard-input.

Requests to the file system are in general order-dependent; if $i > j$, then the response to the ith request may depend on that of the jth request. In the case of the channel system, the nature of the dependencies is dictated by the agents, and in certain cases exhibits *reverse* dependencies. In all cases, external effects may also be felt "between" internal effects. All of this is formalized in [Hea88].

Responses are defined by:

```
data Response = Success
              | Return Contents
              | TagReturn TagContents
              | Failure ErrorMsg

type ErrorMsg = String
```

Thus the response to a request is one of several kinds of success, or failure. `Return` and `TagReturn` occur when results are expected, whereas `Success` occurs when a simple acknowledgement is sufficient. Information about the kind of failure is contained in the `ErrorMsg`, the exact nature of which depends on the request, but is otherwise left unspecified.

The datatypes `Contents` and `TagContents` define the kinds of values that are allowed to be stored in a file or communicated on a channel:

```
type Contents = String
type TagContents = [ (Name,Char) ]
```

A value whose type is an instance of the class `Gap` may be written to a file (or communicated on a channel) by first using `put` to convert it to a string; similarly, to read such a value from a file (or a channel), `get` must be used. [8]

4.2 Continuation-based I/O

HASKELL supports an alternate style of I/O called *continuation-based I/O.* Under this model a HASKELL program is still considered to have type `[Response]->[Request]`, but instead of having

[8]`put` and `get` are automatically derived for each type. For more on this see [Hea88].

the user manipulate the requests and responses directly, a collection of *transactions* are defined which capture the effect of each request/response pair using a continuation style.

Transactions are just functions. For each request `Req` there corresponds a transaction `req`, as shown below: For example, `ReadFile` is a request normally used in a form such as "`ReadFile name`" and is expected to induce either a failure response "`Failure msg`" or success response "`Return contents`". In contrast, using the continuation style the transaction `readFile` would be used in a form such as:

```
readFile name (msg -> error_transaction)
              (contents -> success_transaction)
```

where the second and third arguments are the *failure continuation* and *success continuation*, respectively. If the transaction fails, the error continuation is applied to the error message; if it succeeds the success continuation is applied to the contents of the file. This functionality is defined in terms of requests and responses as shown below:

```
type Behavior    =  [Response] -> [Request]
type SuccCont    =                 Behavior
type RetCont     =  Contents    -> Behavior
type TagRetCont  =  TagContents -> Behavior
type FailCont    =  ErrorMsg    -> Behavior

-- The transactions are:
done          ::                                       Behavior
readFile      :: Name ->          FailCont->RetCont   ->Behavior
writeFile     :: Name ->Contents->FailCont->SuccCont  ->Behavior
appendFile    :: Name ->Contents->FailCont->SuccCont  ->Behavior
deleteFile    :: Name ->          FailCont->SuccCont  ->Behavior
readChannel   :: Name ->          FailCont->RetCont   ->Behavior
readChannels  ::[Name]->          FailCont->TagRetCont->Behavior
appendChannel:: Name ->Contents->FailCont->SuccCont  ->Behavior

done resps = []
readFile name fail succ resps =
   (ReadFile name) :
   case (head resps) of
     Return contents -> succ contents (tail resps)
     Failure msg -> fail msg (tail resps)
writeFile name contents fail succ resps =
   (WriteFile name contents) :
   case (head resps) of
     Success -> succ (tail resps)
     Failure msg -> fail msg (tail resps)
appendFile name contents fail succ resps =
   (AppendFile name contents) :
```

```
   case (head resps) of
     Success -> succ (tail resps)
     Failure msg -> fail msg (tail resps)
deleteFile name fail succ resps =
   (DeleteFile name) :
   case (head resps) of
     Success -> succ (tail resps)
     Failure msg -> fail msg (tail resps)
readChannel name fail succ resps =
   (ReadChannel name) :
   case (head resps) of
     Return contents -> succ contents (tail resps)
     Failure msg -> fail msg (tail resps)
readChannels names fail succ resps =
   (ReadChannels names) :
   case (head resps) of
     TagReturn tcontents -> succ tcontents (tail resps)
     Failure msg -> fail msg (tail resps)
appendChannel name contents fail succ resps =
   (AppendChannel name contents) :
   case (head resps) of
     Success -> succ (tail resps)
     Failure msg -> fail msg (tail resps)
```

4.3 File System Requests

In the descriptions that follow, requests are described using the underlying stream model – the corresponding behavior using the continuation model should be obvious. Also, only the successful situations are described – failures generally result in a system-dependent response indicating the cause of failure. Typical failure messages are `"File not found"`, `"Access rights violation"`, etc.

`ReadFile name`

Accesses the contents of the file named `name`. If successful, the response will be of the form `Return contents`, where the structure of `contents` will depend, of course, on what was written. Typically, and the only required aspect of a valid implementation, the contents will be a (lazy) list of filed characters.

For example, to sum together all of the elements of an integer file (one written with contents `put 0 grade_list ""`) whose name is `"grades"`, one would first issue the request `ReadFile "grades"`. If the response is of the form `Return filed_grade_list`, then:

```
foldl (+) 0 grades
  where (grades, restfile) = (get 0 filed_grade_list)
```

would sum the grades accordingly.

```
WriteFile name contents
```

Associates with the file `name` the contents `contents`. A successful response has form `Success`.

Given the two juxtaposed requests:

```
[ ..., WriteFile name contents1, ReadFile name, ... ]
```

with the corresponding responses:

```
[ ..., Success, Return contents2, ... ]
```

then `contents1 == contents2`, assuming there were no external effects.

```
AppendFile name contents
DeleteFile name
```

These induce the obvious effects, with successful response `Success`.

Note that a proper implementation of `ReadFile` may at times have to make copies of files in order to preserve referential transparency – a successful read of a file should preserve the correct contents, despite future writes to or deletions of the file.

4.4 Channel System Requests

Channels are inherently different from files – they contain "ephemeral" streams of data as opposed to "persistent" stationary values. The most common channels are standard-input, standard-output, and standard-error, and in fact these three are the only required channels in a valid implementation, where they must have the names `"stdin"`, `"stdout"`, and `"stderr"`, respectively.

```
ReadChannel name
```

Opens the channel named `name` for input. The successful response returns the channel contents as a lazy stream. Possible failures include `"Channel does not exist"`, `"Illegal access"`, and `"Channel is write-only"`.

Unlike files, once a channel has been opened, it cannot be opened again in the same program. This reflects the ephemeral nature of its contents and prevents a serious "space leak."

```
ReadChannels [name1, ..., namek]
```

Opens `name1` through `namek` for input. Successful response has type `TagReturn [(Name,Char)]`, where the list of tagged elements is the non-deterministic merge of the individual channels. If an element of this list has form `(namei,ci)`, then it came from channel `namei` in the list of channel names given to `ReadChannels`.

Note that although non-determinism is mentioned, *it is confined to the operating system*, and thus programs using `ReadChannels` are internally referentially transparent.

```
AppendChannel name contents
```

Has the obvious effect of writing `contents` to the channel named `name`. The semantics here is similar to that of `AppendFile` in that the contents is appended to whatever was previously written.

Note that channels cannot be deleted, nor is there a notion of creating a channel.

4.5 Optional Requests

The following requests are not required of a valid HASKELL implementation, but may be useful.

```
CreateProcess prog
```

Has the effect of introducing a new program `prog` into the operating system. `prog` should have the type `[Response] -> [Request]`. This request is necessary if programs such as operating system command line interpreters are to be written in HASKELL.

```
CloseChannel name
```

Closes the named channel which may then be reopened. Certain kinds of devices may require this request.

```
CreateDirectory
DeleteDirectory
```

These induce the obvious effects.

4.6 I/O Semantics

The behavior of a HASKELL program engaged in I/O is given within the context in which it is running. That context will be described using standard HASKELL code augmented with a non-deterministic merge operator.

The state of the operating system (OS state) is completely described by the file system and the channel system. The channel system is split into two subsystems, the input channel system and the output channel system.

```
type State = (FileSystem, ChannelSystem)
type FileSystem    = Name -> Response
type ChannelSystem = (Ics, Ocs)
type Ics    = Name -> (Agent, Open)
type Ocs    = Name -> Response
type Agent = [State] -> Response
type Open   = Pid -> Bool
type Pid    = Int
type Plist = [(Pid,[Request->Response])]
```

Note that an agent maps a list of OS states to responses. Those responses will be used as the contents of input channels, and thus can depend on output channels, other input channels, files, or any combination thereof. For example, a valid implementation is required to allow the user to act as agent between the standard output channel and standard input channel.

Running processes (i.e. programs) are identified by a unique `Pid`. Elements of `Plist` are lists of running programs.

```
os :: TagReqlist -> State -> (TagResplist, State)
type TagResplist = [(Pid,Response)]
type TagReqlist  = [(Pid,Request)]
```

The operating system is modeled as a (non-deterministic) function `os`. The `os` takes a tagged request list and an initial state, and returns a tagged response list, a final state and a list of processes. Given an initial list of programs `start_plist`, `os` must exhibit the following behavior:

```
(tag_resplist, state', plist) = os tag_reqlist state
tag_reqlist = merge [ zip [pid,pid..]
                      (proc (untag pid tag_resplist))
                      || (pid, proc) <- processes ]
processes = start_plist ++ plist
```

where `merge` is a non-deterministic merge of a list of lists, and `untag` is defined below:

```
untag n []                 = []
'     n ((m,resp):resps) = (n==m) => resp:(untag n resps);
                                     untag n resps
```

This relationship accounts for dynamic process creation using `CreateProcess`.

In addition, a valid implementation must ensure that the input channel system is defined at `"stdin"` and the output channel system is defined at both `"stdout"` and `"stderr"`. Furthermore, if the agent attached to standard-input is called `user` (i.e. `ics "stdin"` has form `(user, open)`), then `user` must depend at least on standard-output. In other words, the following constraint must hold:

```
user [..., (fs,(ics,ocs)), ...] = ... user' (ocs "stdout") ...
```

where `user'` is a *strict*, but otherwise arbitrary, function modeling the user. Its strictness corresponds to the user's consumption of standard-output in the process of determining standard-input.

Finally, the required behavior of `os` in response to each kind of request is given in [Hea88].

4.7 Comments

- Since we have already seen that streams can efficiently support other styles of I/O, we chose streams as the primitive in HASKELL (for example, they avoid the linear-space quadratic-time inefficient that would result if continuations were chosen as primitive). This does not mean that streams are the preferred programming model, but just that they are considered simple and general enough to be designated as primitive.
- Note that of the non-deterministic requests only `ReadChannels` is a required feature of HASKELL. This reflects the feeling that while `ReadChannels` is definitely a useful operation, the non-deterministic write operation will find use only in a parallel implementation.
- The only type which the I/O system will handle is `String`. Thus every other type will first have to be converted to/from `String` before/after I/O. This is potentially a problem, for it would be very inconvenient if the programmer had to write two such routines for every type in his program. HASKELL solves this problem by automatically generating such functions for any data type via the derived instance declaration mechanism [Hea88].
- We have introduced the notion of an *agent* that consumes data on output channels and produces data on input channels. This is useful in capturing the semantics of interactive I/O operations. For example, the *user* is an agent who consumes standard output and produces standard input. This particular agent is required to be *strict* in the standard output, corresponding to the notion that the user reads the terminal display before typing at the keyboard.

5 Other Programming Paradigms

What is the relationship between HASKELL and other proposals for concurrent computation? To put the expressive power of HASKELL into context, we will show in this section how other styles can be expressed fairly naturally in HASKELL (or in any functional language incorporating our ideas about functional I/O). The point is not to show "equivalence," but to demonstrate how these styles find a concise expression in HASKELL.

5.1 HASKELL, the Actor

The Actor Model of computation [Agh86] consists of a number of computational agents called *actors*. An actor maps each incoming communication to a 3-tuple consisting of:

- A finite set of communications sent to other actors.

- A replacement behavior (which governs the response to the next communication processed).
- A finite set of new actors created.

It is not difficult to see how the HASKELL I/O system can be used to write actor programs. Each actor is represented by a HASKELL program. In the actor model, communication is handled by an underlying invisible mail system (similar in ways to Stoye's sorting office) — in HASKELL one I/O channel can be dedicated to function as this mailbox. Note that each process needs a unique tag to identify addressees of the messages. Thus, communication can occur by I/O write operations. The replacement behavior is simply expressed as a recursive function call on the rest of the messages for the actor. Creating new actors can be achieved via HASKELL's **CreateProcess**.

There have been several languages based on the actor model. Agha [Agh84] defines one called SAL. The following SAL program fragment models a shared bank account. (It is a simplified version of the example in [Agh84].) Only two operations, namely depositing and balance querying, are allowed. Both these operations return the balance after completion. (In the program that follows, **r** specifies the type of the transaction, **d** the amount involved, and **m** specifies the mail address of the customer.)

```
def bank_account (a) [r, d, m]
   if r = deposit then
      become bank_account <a+d> //
      send [a+d] to m
   fi
   //
   if r = balance then
      become bank_account <a> //
      send [a] to m
   fi
end def
```

The corresponding HASKELL program would look as follows. The program can be made to look even more like the actor version by defining syntax to mimic *become, send* and other SAL primitives.

```
main resps =
  [ ReadChannel "mailsystem",
    WriteChannel "mailsystem" answers ]
    where answers = put (bank_account 0 queries)
          queries = get resps!!1
          bank_account a [] = []
          '            a (DEPOSIT d m : rest) = (a+d,m) : bank_account (a+d) rest
          '            a (BALANCE m   : rest) = (a,  m) : bank_account a     rest
```

5.2 The UNITY in HASKELL

UNITY is a computational model and a proof system [CM88] for developing programs in general and parallel programs in particular. A UNITY program consists of a declaration of variables, a specification of their initial values, and a set of multiple assignment statements. A program execution starts from any state satisfying the initial condition and goes on forever; in each step of execution some assignment statement is selected non-deterministically and executed. Non-deterministic selection is constrained by the following "fairness" rule: Every statement is selected infinitely often.

The following UNITY program (from [CM88]) non-deterministically merges two sequences x and y into z.

```
Program MUX
  assign
      x,z := tail(x), z;("x",head(x)) if x != null
      y,z := tail(y), z;("y",head(y)) if y != null
end {MUX}
```

In HASKELL, this could be naturally expressed as follows: The state of the program would be represented as a tuple of the relevant values, and each of the UNITY multiple-assignment statements would be written as a state transforming function. The non-deterministic control can be provided by the oracle form of non-determinism described before. At each stage, one of the functions would be chosen by inspecting the ressult of a `choose` transaction, and applied to the state to obtain a new state.

```
mux (x,y,[]) where
             mux state = choose [f1, f2] (f ->
                               mux (f state))
             f1 (x:xs, y, z) = (xs, y, z ++ [x])
             f2 (x, y:ys, z) = (x, ys, z ++ [y])
```

This style of programming in HASKELL has been used previously to express asynchronous process simulations [HA88].

5.3 HASKELL meets Linda

Linda [Gel85] is a parallel programming language, where a program is a set of cooperating processes communicating asynchronously via a global data structure called *tuple space*. Processes add tuples to tuple space by an `out` operation. The operation of removing tuples from tuple space (`in`) takes a template and instantiates variable slots with values from a tuple which matches the constant slots.

HASKELL already provides a process oriented model (the only difference being that the processes must be HASKELL programs). The flexibility of the I/O system can be put to good use by providing a I/O device called (say) `tuplespace`. The operations on this tuple space are in the form of requests corresponding to `in` and `out`.

A common Linda paradigm is the "task bag" paradigm, where processes pick out tasks from a bag, and add in new tasks generated. In Linda, each process would look as follows:

```
process_i
{
  do forever {
    in ("task", var task-descriptor) /*pick task to work on*/
      ... process the task ...
    out("task", task-descriptor)     /*add new task to bag */
  }
}
```

In HASKELL this would look as follows:

```
process_i resps = [ReadChannel "tuplespace"
                        ("task", var task-descriptor),
                   WriteChannel "tuplespace" new_tasks]
                   where new_tasks = process tasks
                         tasks = resps!!1
                         ... description of process ...
```

5.4 Communicating Sequential (HASKELL) Processes

Much work has been done in developing a calculus of interacting processes, the two most important being CCS [Mil81] and CSP [Hoa78,Hoa85]. It is clear that the parallel programming paradigm CSP has a communication structure which could be reproduced in HASKELL. The main difference, as Hoare points out in a comparison with [Kah74], is that communication in CSP is synchronous, while functional multiprogramming models (including HASKELL) are asynchronous. To force synchronous execution of HASKELL processes, the "sorting office" described earlier could match input and output requests before sending out a response. Turner [Tur87] refines Stoye's scheme to achieve synchronous communication in a similar way.

In [Hoa85] Hoare also points out that in a functional multiprocessing system the processes themselves are deterministic; for example it is not possible for a process to wait for the first of two inputs. This criticism clearly does not apply to HASKELL, since non-deterministic requests like `ReadChannels` give us the desired functionality without destroying referential transparency.

6 Conclusion

Contrary to popular belief, purely functional I/O can be both flexible and concise. Lazy evaluation, one of the most important tools in the functional programmer's toolbox, serves us well in the context of I/O; perhaps this should come as no surprise. We plan to gain more practical experience by writing numerous functional programs involving I/O using HASKELL.

Non-determinism is manifested in the "glue" with with programs are put together. Thus the programs themeselves remain referentially transparent. These approaches also naturally apply to work on writing non-determinate systems in a functional language (notably, functional operating systems).

References

[Agh84] Gul Agha. *Semantic Considerations in the Actor Paradigm of Concurrent Computation*, pages 151–179. Volume 197 of *Lecture Notes in Computer Science*, Springer Verlag, 1984.

[Agh86] Gul Agha. *Actors A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.

[BMS80] Burstall, McQueen, and Sanella. Hope: an experimental applicative language. In *Proceedings 1st International LISP conference, Stanford*, 1980.

[Bur88] F. W. Burton. Nondeterminism with referential transparency in functional programming languages. *The Computer Journal*, 31(3):243–247, 1988.

[BWW86] John Backus, John H. Williams, and Edward L. Wimmers. *FL Language Manual (Preliminary Version).* Technical Report RJ 5339 (54809), IBM Almaden Research Center, November 1986.

[Cli82] W. Clinger. Nondeterministic call by need is neither lazy nor by name. In *Proc. 1982 ACM Symp. LISP and Functional Programming*, 1982.

[CM88] K. Mani Chandy and Jayadev Mishra. *Parallel Program Design.* Addison-Wesley, 1988.

[Fai85] Jon Fairbairn. *Design and Implementation of a Simple Typed Language Based on the Lambda-Calculus.* Technical Report 75, University of Cambridge Computer Laboratory, May 1985.

[Gel85] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985.

[HA88] Paul Hudak and Steve Anderson. *Haskell Solutions to the Language Session Problems at the 1988 Salishan High-Speed Computing Conference.* Technical Report YALEU/DCS/RR-627, Yale University, Department of Computer Science, January 1988.

[Hea88] Paul Hudak and Philip Wadler et al. *Report on the Functional Programming Language Haskell.* Technical Report YALEU/DCS/RR-666, Department of Computer Science, Yale University, December 1988.

[Hen82] P. Henderson. Purely functional operating systems. In J. Darlington, P. Henderson, and D.A. Turner, editors, *Functional Programming and its Applications*, pages 177–189, Cambridge University Press, 1982.

[HMT88] Robert Harper, Robin Milner, and Mads Tofte. *The Definition of Standard ML Version 2.* Technical Report ECS-LFCS-88-62, Laboratory for Foundations of Computer Science, Department of Computer Science - University of Edinburgh, August 1988.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Comm. ACM*, 21(8):666–677, 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Hol83] Sören Holmström. *PFL: A Functional Language for Parallel Programming, and its Implementation.* Technical Report 7, Programming Methodology Group, University of Göteborg and Chalmers University of Technology, September 1983.

[Hud84] Paul Hudak. *ALFL Reference Manual and Programmer's Guide.* Technical Report YALEU/DCS/TR-322, Yale University Department of Computer Science, August 1984.

[Kah74] G. Kahn. *Information Processing*, chapter The Semantics of a simple language for Parallel Programming, pages 471–475. Volume 74, North Holland, 1974.

[Kar81] K. Karlsson. *Nebula, a Functional Operating System.* Technical Report, Chalmers University, 1981.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[Lan65] P. J. Landin. A correspondence between algol 60 and church's lambda notation. *Comm. ACM*, 21(11):931–933, 1965.

[MH88] Lee M. McLoughlin and Sean Hayes. Interlanguage working from a pure functional language. Functional Programming mailing list, November 1988.

[Mil81] Robin Milner. *A Calculus of Communicating Systems.* Volume 81 of *Lecture Notes in Computer Science*, Springer Verlag, 1981.

[SS88] Harald Sondergaard and Peter Sestoft. *Nondeterminism in Functional Languages.* Technical Report 88/18, Department of Computer Science, University of Melbourne, 1988.

[Sto84] W. Stoye. *A New Scheme for Writing Functional Operating Systems.* Technical Report 56, Cambridge University Computer Laboratory, 1984.

[Tur85] David Turner. Miranda: a non-strict functional language with polymorphic types. In *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architecture, Nancy France (Springer Lecture Notes in Computer Science, vol 201)*, September 1985.

[Tur87] David Turner. *Functional Programming and Communicating Processes*, pages 54–74. Volume 259 of *Lecture Notes in Computer Science*, Springer Verlag, 1987.

[WW88] John H. Williams and Edward L. Wimmers. Sacrificing simplicity for convenience: where do you draw the line? In *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages , San Diego, California*, January 1988.